

Tomasz "Zyx" Jędrzejewski

Piszemy chatbota

Wersja 1.0 (12.09.2007)



Szczegółowe informacje o licencji znajdują się pod artykułem.

www.zyxist.com

Piszemy chatbota

Chatbot to program komputerowy, z którym można prowadzić rozmowę w języku naturalnym przy użyciu tekstowego interfejsu, najczęściej stworzony, aby symulować prawdziwego człowieka. Aplikacje tego rodzaju powstają już od lat pięćdziesiątych ubiegłego wieku, lecz dopiero wzrost wydajności komputerów w ostatnich latach umożliwił im rozwinięcie skrzydeł. Jednak wbrew pozorom stworzenie własnego programu konwersacyjnego, który w dodatku potrafiłby się uczyć, nie jest takie trudne. Postaram się pokazać to w niniejszym artykule.

Test Turinga

Gdy powstały pierwsze komputery, rozpoczęła się debata, czy potrafią one naśladować prawdziwą inteligencję. Jeden z pionierów informatyki, Alan Turing zaproponował w 1950 roku test mający pośrednio udowodnić zdolności maszyn do posługiwania się językiem naturalnym w sposób symulujący człowieka. Od jego nazwiska zwany jest właśnie „testem Turinga”. Polega on na rozmowie człowieka – sędziego z kilkoma rozmówcami w języku naturalnym. Jeśli po jej zakończeniu nie jest w stanie określić, która ze stron jest maszyną, wtedy mówi się, że maszyna taka przeszła test. Oczywiście sędzia wie, że wśród stron rozmowy jest jakiś program komputerowy i na podstawie tego może. Proste programy konwersacyjne działające losowo są w stanie oszukać część ludzi przynajmniej przez krótki czas, jeśli nie mają oni podstaw, by podejrzewać, że po drugiej stronie może znajdować się bot.

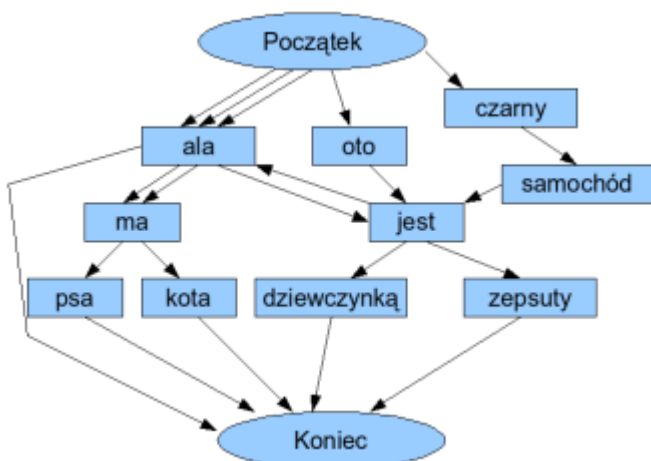
Turing ocenił, że około roku 2000 maszyny z pamięcią ok. 10^9 bitów będą w stanie oszukać do 30% sędziów, jednak jak pokazuje rzeczywistość, był nastawiony do eksperymentu dość optymistycznie. Jak dotąd żaden program nie sprostował zadaniu. Co roku organizowane są różne konkursy mające na celu przejście testu. Najbardziej prestiżowym jest doroczna Nagroda Loebnera organizowana w *The Cambridge Center of Behavioral Studies*, w której wyróżnia się najlepsze programy konwersacyjne. Do wygrania jest wiele cennych rzeczy. Autor programu, który jako pierwszy oszuka wszystkich sędziów otrzyma medal wykonany z 18-karatowego złota oraz 100 000 dolarów. Autor programu, który wprowadzi testu nie przeszedł, ale zdaniem sędziów najlepiej naśladował inteligencję, otrzymuje natomiast brązowy, pozłacany medal oraz 2 000 USD. Jest więc o co walczyć, jednak żeby doprowadzić swój program do takiego poziomu, potrzeba dużo pracy.

Zasada działania

Zaproponowany tutaj algorytm działania można nazwać trywialnym, zaś na początku trudno uwierzyć, że on faktycznie działa. Znalazł go kilka lat temu w sieci mój znajomy, a następnie podpiął oparty o niego program do Gadu-Gadu, nadając mu imię, wiek oraz kilka podstawowych informacji. Przeglądając logi rozmów można było łatwo dostrzec sporą grupę ludzi, którzy autentycznie dawali się nabrać i sądzili, że rozmawiają z człowiekiem, aczkolwiek nieco ześwirowanym. Przejdźmy jednak do spraw technicznych.

Nasz chatbot będzie utrzymywać w pamięci graf skierowany, w którym wierzchołki reprezentują

słowa, zaś krawędzie – połączenia między nimi. Graf jest automatycznie aktualizowany w trakcie dyskusji poprzez wbudowywanie do niego fraz wypowiedzianych przez ludzkiego rozmówcę. Wyróżnione są w nim dwa wierzchołki, które oznaczymy \$ i # - odpowiednio początek i koniec wypowiedzi. Generowanie odpowiedzi polega na wybraniu losowej ścieżki pomiędzy \$ oraz #, a następnie połączeniu leżących w jej wierzchołkach słów za pomocą spacji. Pokażmy to wszystko na przykładzie. Załóżmy, że w rozmowie powiedzieliśmy następujące zdania:



1. ala ma kota
2. ala ma psa

3. *ala jest dziewczynką*
4. *czarny samochód jest zepsuty*
5. *oto jest ala*

W rezultacie otrzymamy graf podobny do tego na rysunku obok. Zauważmy, że obecne są w nim krawędzie wielokrotne: potrójna do słowa „ala” oraz podwójna z „ala” do „ma”. Jest to wynikiem tego, że słowo „ala” znalazło się trzy razy na początku wypowiedzi, zaś „ma” dwukrotnie pojawiało się po słowie „ala”. Działający losowo program może równie dobrze uzyskać jedno ze zdań wpisanych przez użytkownika, jak i wymieszać ze sobą ich fragmenty, przykładowo otrzymując „czarny samochód jest dziewczynką” lub „ala jest zepsuty”.

Implementacja

Do implementacji na potrzeby artykułu wykorzystałem stosunkowo niedawno wydany język D. Osoby znające C++ nie powinny mieć jednak problemów z przerobieniem go na swój język oraz bibliotekę STL. Graf reprezentowany jest przez zmodyfikowaną listę sąsiedztwa; mianowicie główny spis wierzchołków to tablica haszująca, w której indeksem są poszczególne słowa. Dzięki temu możliwe będzie szybkie sprawdzenie, czy dany wyraz posiada już w bazie, czy nie. Lista sąsiedztwa w każdym wierzchołku przechowuje natomiast wskaźniki do obiektów, dzięki czemu po wylosowaniu jednego z nich nie trzeba znów sięgać do tablicy, aby odnaleźć obiekt. Ponieważ w trakcie działania programu nie są usuwane z grafu żadne wierzchołki ani krawędzie, nie narażamy się na niespójność danych. Dodatkowo każdy wierzchołek otrzymuje unikalny numer ID, który przyda się przy zapisywaniu i odczytywaniu istniejącego słownika z pliku.

Kodowanie zaczynamy od załączenia kilku przydatnych modułów:

```
import std.stdio;          // standardowe wejście/wyjście
import std.string;        // obsługa ciągów
import std.random;        // liczby losowe
import std.stream;        // strumienie, obsługa plików
import std.conv;          // funkcje konwersji
import std.c.time;        // czas do zainicjowania generatora losowego

ulong nodePtr = 0;        // generator ID dla wierzchołków
```

Klasa *wordNode* reprezentować będzie pojedynczy wierzchołek grafu – słowo:

```
class wordNode
{
    private wordNode nodeList[ulong];
    private ulong counter = 0;
    private ulong id;
    private char[] word;
```

Mamy tutaj kilka pól. Pierwsze z nich, *nodeList* to lista wszystkich wierzchołków grafu, z którymi dane słowo jest połączone. Do jej implementacji wykorzystana jest tablica asocjacyjna z numerycznymi indeksami, dzięki czemu możemy prosto dodawać do niej nowe elementy. Pole *counter* służy do nadawania nowym połączeniom kolejnych indeksów. *id* jest globalnym, numerycznym identyfikatorem słowa przydatnym podczas odczytywania słownika z pliku. Na końcu mamy *word* zawierające po prostu podane słowo. Tutaj również jest ono niezbędne, ponieważ przy losowaniu ścieżki nie korzystamy już z głównej tablicy haszującej, lecz przeskakujemy bezpośrednio z wierzchołka do wierzchołka i musimy mieć prosty sposób sprawdzenia, jaki wyraz on reprezentuje.

Klasa będzie posiadać dwa konstruktory. Jeden będzie wykorzystywany do uzupełniania bazy w trakcie rozmowy, zaś drugi przy wczytywaniu słownika z pliku – umożliwi bowiem ręczne określenie ID. Dodatkowo zadeklarujemy dwie metody dostępowe: *getWord()* oraz *getId()*:

```
this(char[] _word)
{
    id = nodePtr++;
    word = _word;
} // end this();

this(char [] _word, ulong _id)
```

```

    {
        id = _id;
        word = _word;
    } // end this();

    public char [] getWord()
    {
        return word;
    } // end getWord();

    public ulong getId()
    {
        return id;
    } // end getId();

```

Czas nadać naszej klasie jakąś funkcjonalność. Metoda *chooseNext()* będzie zwracać losowo wybrany kolejny wierzchołek grafu, z którym dane słowo ma przynajmniej jedno połączenie – zatem w tym miejscu dokonuje się „decyzja”, co bot dokładnie powie człowiekowi :). Ponadto mamy też metodę *throwConnections()*, która będzie wykorzystywana przy rzucaniu zawartości słownika programu do pliku. Otrzyma ona obiekt pliku i będzie musiała zapisać do niego w jednym wierszu najpierw swój własny ID, a później identyfikatory wszystkich następników odseparowane przecinkiem. Całość musi być zakończona dla pewności znakiem **#**. Natomiast na potrzeby wczytywania słownika oraz uczenia się, została napisana metoda *addConnection()* dodająca nowe połączenie między dwoma wyrazami:

```

public wordNode chooseNext()
{
    if(nodeList.length == 0)
    {
        return null;
    }
    return nodeList[rand() % nodeList.length];
} // end chooseNext();

public void throwConnections(File f)
{
    wordNode n;
    f.printf("%d,", id);
    foreach(n; this.nodeList)
    {
        f.printf("%d,", n.getId());
    }
    f.writefln("#");
} // end throwConnections();

public void addConnection(wordNode n)
{
    this.nodeList[this.counter++] = n;
} // end addConnection();
} // end wordNode;

```

rand() to wolniejszy, ale za to dokładniejszy wariant znanej z języka C funkcji o tej samej nazwie. Zwraca on losowy ciąg 32 bitów bez znaku. Dzieląc otrzymany wynik modulo przez ilość połączeń, otrzymujemy liczbę losową z przedziału 0 do *nodeList.length*. Podobnie jak w C++, także i w D można reprezentować pliki poprzez strumienie, korzystając z dokładnie tych samych funkcji, których używa się do wypisywania tekstu na konsolę. Dodatkowo, dzięki pętli *foreach* możemy łatwo pobrać wszystkie elementy naszej listy sąsiedztwa. Oczywiście kompilator sam użyje do jej implementacji wskaźników; my nie musimy się o to troszczyć.

Teraz będzie trochę strukturalnego kodu. Musimy napisać sobie trzy funkcje. Pierwsza z nich to *tokenize()*, która zwróci tekst wpisany przez użytkownika rozbity na poszczególne słowa oraz pozbawiony wszelkich znaków interpunkcyjnych, których nie chcemy zapamiętywać w naszej bazie. Dwie kolejne: *dictionaryLoad()*

oraz *dictionarySave()* służą do odczytywania i zapisywania słownika do pliku tekstowego tak, aby efekty nauki nie były gubione po wyłączeniu programu.

```
char [][][int] tokenize(char [] src)
{
    int cnum = 1;
    int i;

    char forbidden[] = ",./:;'[]\?><\"{}()_~!@#$$%^&*` \t\n\r";
    char [][][int] result;
    bool prevForbidden = false;
    bool isForbidden = false;
    result[0] = "$";
    for(i = 0; i < src.length; i++)
    {
        int j;
        forbiddenCheck:for(j = 0; j < forbidden.length; j++)
        {
            // Ignorujemy znaki „forbidden” oraz sterujące
            if(src[i] == forbidden[j] || src[i] < 32)
            {
                isForbidden = true;
                break forbiddenCheck;
            }
        }

        if(isForbidden)
        {
            // Mamy niedozwolony znak, zapamiętujemy to
            prevForbidden = true;
        }
        else
        {
            if(prevForbidden)
            {
                // Poprzednio był niedozwolony znak, zainicjuj
                // więc nowe słowo
                result[++cnum] = "";
            }
            prevForbidden = false;
            result[cnum] += src[i];
        }
        isForbidden = false;
    }
    result[++cnum] = "#";
    return result;
} // end tokenize();
```

Dzielenie tekstu na słowa z jednoczesnym usuwaniem znaków nie jest przesadnie skomplikowane. Jedziemy sobie pętlą po wpisanej frazie. Jeśli napotkamy znak zabroniony, nie wpisujemy go do bufora, tylko przeskakujemy do kolejnego. Jedynie zapamiętujemy, że napotkaliśmy właśnie coś takiego, a to dlatego, że gdyby następną była jakaś litera (czyli treść dozwolona), powinniśmy wtedy rozpocząć nowy wyraz. Tutaj taka mała uwaga – aby oszczędzić sobie dodatkowych instrukcji warunkowych w kolejnym etapie, już tutaj dodajemy do tablicy słów wyrazy specjalne, czyli \$ i #.

Słownik zapisany będzie w pliku w następującym formacie: na początku znajduje się X wierszy reprezentujących informacje o poszczególnych słowach (czyli ID numeryczny i słowo oddzielone od siebie przecinkiem). Później jest linijka kontrolna ze znakiem \$, po czym idą opisy połączeń w grafie między wierzchołkami. Każda linijka opisuje połączenia wychodzące z pojedynczego wierzchołka, a są one reprezentowane przez listę ID oddzielonych przecinkami. Pierwszy ID określa numer wierzchołka źródłowego, później idzie ileś tam wierzchołków i zakończenie wykazu w postaci znaku #. Przykładowy plik

ma postać:

```
0,$
1,#
2,ala
3,ma
4,kota
5,psa
$
0,2,#
2,3,#
3,4,5,#
4,1,#
5,1,#
```

Kod do zapisywania danych w tym formacie prezentowany jest poniżej. W tym miejscu musimy także zadeklarować sobie tablicę z haszowaniem do przechowywania naszego grafu (allNodes). Dzięki kluczom słownym będzie możliwy szybki dostęp do poszczególnych wierzchołków. Zmienna state przyda się później, przy tworzeniu interfejsu rozmowy.

```
wordNode [char[]] allNodes;
byte state;

int dictionarySave(char[] filename)
{
    try
    {
        File dictFile = new File(filename, FileMode.Out);
        // Zrzuc liste slow
        wordNode n;
        foreach(n; allNodes)
        {
            dictFile.writefln(std.string.toString(n.getId())~", "~n.getWord());
        }
        // Zrzuc liste polaczen
        dictFile.writefln("$");
        foreach(n; allNodes)
        {
            n.throwConnections(dictFile);
        }
        dictFile.close();
        return 1;
    }
    catch(OpenException)
    {
        writefln("Wystapil blad podczas zapisywania slownika do pliku
"~filename);
    }
    return 0;
} // end dictionarySave();
```

Kod do odczytywania jest nieco dłuższy:

```
int dictionaryLoad(char [] filename)
{
    try
    {
        File dictFile = new File(filename, FileMode.In);

        wordNode [ulong] db;
        byte readState = 0;
        char [] line;
        ulong maxNodep = 0;
```

```

while(!dictFile.eof())
{
    // Podziel wczytana linie wzgledem przecinkow
    char[][] lineData = std.string.split(dictFile.readLine(),
",");

    if(readState == 0)
    {
        // Odczyt listy slow
        if(lineData[0] == "$")
        {
            readState = 1;    // Zmiana stanu
            continue;
        }
        ulong wid = std.conv.toUlong(lineData[0]);
        db[wid] = allNodes[lineData[1]] = new
wordNode(lineData[1], wid);
        if(wid > maxNodep)
        {
            maxNodep = wid;
        }
    }
    else
    {
        // Odczyt listy polaczen miedzywierzcholkowych
        ulong wid = std.conv.toUlong(lineData[0]);
        ulong i, j;
        for(i = 1; i < lineData.length; i++)
        {
            if(lineData[i] == "#")
            {
                // Koniec listy polaczen dla danego
                // wierzcholka
                break;
            }

            // przekonwertuj ID na liczbe i dodaj polaczenie
            j = std.conv.toUlong(lineData[i]);
            db[wid].addConnection(db[j]);
        }
    }
}
// Zainicjuj poprawnie generator ID wierzcholkow
nodePtr = maxNodep + 1;
dictFile.close();
}
catch(OpenException)
{
    writeln("Wystapil blad podczas odczytywania slownika z pliku
"~filename);
}
return 0;
} // end dictionaryLoad();

```

Ładowarka oparta jest o system stanów. Gdy pętla znajduje się w stanie 0, program wczytuje spis wierzchołków. Napotkanie symbolu \$ oznacza przejście do stanu 1 oznaczającego odczyt połączeń między wierzchołkami. Na czas wczytywania utrzymujemy dodatkowy spis wszystkich wierzchołków (zmienna *db*) oparty o ich numeryczne indeksy. Potem nie jest on już jednak potrzebny, dlatego pozostawiamy go na pastwę odśmieczacza pamięci. W trakcie odczytywania musimy również odtworzyć wartość zmiennej *nodePtr*, tak aby dodane w trakcie kolejnej sesji słowa otrzymały nowe indeksy numeryczne, a nie nadpisały stare.

Pora wziąć się za funkcję *main()*. Program będzie mógł obsłużyć do trzech parametrów uruchomienia:

1. Plik słownika – jeśli takowy nie jest podany, używany jest *slovník.dict*. Jeśli podamy nazwę

nieistniejącego pliku, program zainicjuje pusty słownik.

2. Ksywka użytkownika (domyślnie „User”)
3. Ksywka komputera (domyślnie „Komputer”)

Dodatkowo historia rozmowy będzie zapisywana do pliku, aby można było później się pośmiać wspólnie ze znajomymi :).

```
int main(char[][] arg)
{
    // Wczytaj słownik
    char [] dictName = "słownik.dict";
    char [] userNick = "User";
    char [] botNick = "Komputer";

    switch(arg.length)
    {
        case 4:
            botNick = arg[3];
        case 3:
            userNick = arg[2];
        case 2:
            dictName = arg[1];
    }

    if(std.file.exists(dictName))
    {
        dictionaryLoad(dictName);
    }
    // Zainicjuj generator liczb losowych
    time_t r = time(NULL);
    rand_seed(cast(uint)r, 0);

    // Zainicjuj historie rozmow
    File logs = new File("chatlog.log", FileMode.Append);
    logs.writefln("-----Nowa sesja ze słownikiem "~dictName~"-----");
}
```

Teraz pora na część rozmawiającą. Rozmowa będzie składać się z zapętlonych dwóch faz. Na początku użytkownik wpisuje jakiś tekst, który jest następnie zapamiętywany w grafie. W kolejnym kroku program generuje własną, losową odpowiedź. Kod fazy pierwszej prezentuje się następująco:

```
// Gadaj
while(1)
{
    if(state == 0)
    {
        // Oczekiwanie na tekst użytkownika
        writef("[~userNick~]: ");
        char cmd[] = readln();

        // Jesli wpisal komende sterujaca,
        if(std.string.find(cmd, "\\quit") == 0)
        {
            break;
        }
        if(std.string.find(cmd, "\\help") == 0)
        {
            writefln("--- Zyxobot v. 0.0.1 ---");
            writefln("Bot do rozmowy z komputerem z artykulu
'Piszemy chatbota'.");
            writefln("Autor: Zyx [http://www.zyxis.com]");
        }
    }
}
```

```

        writefln("");
        writefln("wpisz \\quit aby wyjść");
        continue;
    }
    logs.writefln("[~userNick~]: ~cmd);
    // Podziel wypowiedz na slowa
    char [][]int] words = tokenize(cmd);
    ulong i;

    // Zapamietaj wypowiedz w grafie
    for(i = 0; i < words.length; i++)
    {
        words[i] = std.string.toLowerCase(words[i]);
        if((words[i] in allNodes) == null)
        {
            allNodes[words[i]] = new wordNode(words[i]);
        }
        if(i > 0)
        {
            allNodes[words[i-1]].addConnection(allNodes[words[i]]);
        }
    }
    // Przejdź do stanu reakcji
    state = 1;
}

```

Na tym etapie możemy też rozpoznać dwie komendy sterujące: **\help** do wyświetlenia tekstu pomocno-informacyjnego oraz **\quit** kończącego sesję. Nie są one interpretowane jako część wypowiedzi. Zauważmy, że dzięki tablicom haszującym możemy szybko sprawdzić, czy jakieś słowo już ma swój wierzchołek. Dzięki uwzględnieniu specjalnych słów początkowych i końcowych już na etapie dzielenia na tokeny, a także nadanie początkowej inicjatywy użytkownikowi gwarantuje, że nawet pusty słownik zostanie na początku zainicjowany prawidłowo, a także że w takim wypadku program będzie miał czym odpowiedzieć.

Przejdźmy do odpowiedzi programu:

```

else
{
    // Reakcja programu.
    // Ustaw się na węzle początkowym
    wordNode n = allNodes["$"];
    writefln("[~botNick~]: ");
    logs.writefln("[~botNick~]: ");

    // Losuj kolejne słowa, dopóki nie dojdiesz do wierzchołka
koncowego
    while(1)
    {
        n = n.chooseNext();
        if(n.getWord() == "#")
        {
            break;
        }
        writefln(n.getWord() ~ " ");
        logs.writefln(n.getWord() ~ " ");
    }
    writefln("");
    logs.writefln("");
    state = 0;
}

```

Losowanie słów to prosta pętla pobierająca losowo kolejne wierzchołki, dopóki nie trafi na końcowy. Po wyświetleniu z powrotem ustawiamy stan oczekiwania na użytkownika. Końcówka programu to dodanie

tekstu pożegnalnego do logów i zapisanie stanu słownika:

```
    }
    logs.writefln("-----[koniec sesji]-----");
    logs.close();
    dictionarySave(dictName);
    return 0;
} // end main();
```

To wszystko, jeśli chodzi o kod.

Użycie

Program jest prosty w użyciu. Kompilujemy go poleceniem:

```
dmd chatbot.d
```

Następnie uruchamiamy:

```
./chatbot slownik.dict ksywkaUsera ksywkaKomputera
```

I rozmawiamy. Przy pierwszym uruchomieniu musi minąć trochę czasu, zanim słownik wypełni się na tyle, że rozmowa zacznie przypominać rozmowę, a nie dosłowne powtarzanie tekstów. Na wszelki wypadek zalecam pisanie bez polskich znaków – D pracuje domyślnie w unikodzie, a jeśli w konsoli mamy inne kodowanie, mogą być problemy, których rozwiązania jeszcze nie znalazłem.

Program może dostarczyć całkiem dużo zabawy, szczególnie przy większej grupie osób. Oto fragmenty z moich własnych rozmów na słowniku zainicjowanym tematami politycznymi oraz antymicrosoftowymi:

```
[Komputer]: dyskusja z windowsem bym szybko skapitulowal
[User]: z windowsem nie da sie rozmawiac za to z zyxem i owszem
[Komputer]: chyba ze jestem komputerem
[User]: nawet komputer nie potrafi rozmawiac z windowsem
[Komputer]: oczywiscie
[User]: nikt nie potrafi rozmawiac z tym systemem
[Komputer]: sprawdzmy czy pis i pozytecznosc
[User]: ok, masz jakiegos pisowca tu gdzie?
[Komputer]: nawet komputer nie potrafi rozmawiac za niego placic slone
pieniadze nie kloc sie rozmawiac
-----
[Eniac]: krakow wydaje sie byc optymalnym miejscem do wszystkich diablów
[Zyx]: ile juz znasz slowek, co?
[Eniac]: sprawiles wlasnie wrazenie bycia inteligentnym
```

Niestety, przy bardzo dużym słowniku i dużej ilości różnorodnych połączeń odpowiedzi programu zaczynają coraz bardziej przypominać bełkot. Spróbowałem to jakoś ograniczyć i wpadłem na pomysł, by oddzielić od siebie listę połączeń między pierwszym a drugim wyrazem wypowiedzi od pozostałych. Dzięki temu słowo „muszę” wypowiedziane jako pierwsze rozpoznawane jest jako coś innego, niż gdy znajduje się w środku długiej wypowiedzi. Okazuje się, że w ten sposób program może prosto unikać generowania bezsensownych składanek, a otrzymywane rezultaty są znacznie bardziej porażające. Poniżej kilka próbek z ulepszonej wersji chatbota:

```
[Eniac]: na polityczna emeryture
[Zyx]: chcesz isc na emeryturke?
[Eniac]: platforma powinna tym razem wygrac
[Zyx]: aaa, pis mial isc na emeryture.
[Eniac]: mysle wiec jestem komputerem
-----
[Zyx]: windows xp byl lepszy od visty
[Eniac]: microsoft ssie
[Zyx]: microsoft ciagnie grube pieniadze na monopolu
[Eniac]: ale to wrog polski
-----
[Eniac]: czy wiesz ze kaczynski nie wie co robi dla tego bylo malo
powiedziane
```

```
[Zyx]: nie zgadzam sie!  
[Eniac]: musisz wiedziec ze linux jest czlowiekiem  
-----  
[Zyx]: skalny dom zional pustka  
[Eniac]: odpowiadasz inteligentniej  
[Zyx]: odpowiadasz nieglupio  
[Eniac]: musisz wiedziec ze moze przejsc test turinga  
-----  
[Zyx]: jestem sobie botem i jestem napisany w D  
[Eniac]: ja jestem czlowiekiem z drzew  
-----  
[Zyx]: krakow lezy w polsce  
[Eniac]: jestem sobie botem i wladza to linux
```

Zakończenie

Tworzenie botów z prawdziwego zdarzenia wymaga znacznie więcej finezji. Aby generowały poprawne składniowo wypowiedzi, należy nauczyć je gramatyki oraz wbudować w nie kilka dodatkowych wynalazków umożliwiających prawidłowe reagowanie na kontekst wypowiedzi. Jednak od czegoś trzeba zacząć, a taki prosty chatbot świetnie pokazuje, że podstawowa idea opiera się na nieskomplikowanym algorytmie. Pierwotna wersja powyższego kodu, bez dodatkowych wynalazków w stylu zapamiętywania słownika, zajmowała jedynie 2 kilobajty kodu. Może to być świetny punkt wyjścia do dalszej zabawy z chatbotami.

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na **www.zyxist.com**

Licencja

Artykuł rozpowszechniany jest na licencji **Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 2.5 Polska**.

Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór

Na następujących warunkach:

1. *Uznanie autorstwa*. Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę*.
2. *Użycie niekomercyjne*. Nie wolno używać tego utworu do celów komercyjnych.
3. *Bez utworów zależnych*. Nie wolno zmieniać, przekształcać ani tworzyć nowych dzieł na podstawie tego utworu.

W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór.

Każdy z tych warunków może zostać uchylony, jeśli uzyska się zezwolenie właściciela praw autorskich.

Powyższe postanowienia w żaden sposób nie naruszają uprawnień wynikających z dozwolonego użytku ani żadnych innych praw.

Internetowa skrócona wersja licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/>

Pełen tekst licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/legalcode>

Podczas publikacji należy podać następujące informacje:

1. Link do internetowej skróconej wersji licencji.
2. Informację o wersji publikowanego tekstu.
3. Informacje o autorze w następujący sposób:

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na **www.zyxist.com**